TwigList: Make Twig Pattern Matching Fast

Lu Qin, Jeffrey Xu Yu, and Bolin Ding

The Chinese University of Hong Kong, China
{lqin,yu,blding}@se.cuhk.edu.hk

Abstract. Twig pattern matching problem has been widely studied in recent years. Give an XML tree $\mathcal{T}$. A twig-pattern matching query, Q, represented as a query tree, is to find all the occurrences of such twig pattern in $\mathcal{T}$. Previous works like *HolisticTwig* and *TJFast* decomposed the twig pattern into single paths from root to leaves, and merged all the occurrences of such path-patterns to find the occurrences of the twig-pattern matching query, Q. Their techniques can effectively prune impossible path-patterns to avoid producing a large amount of intermediate results. But they still need to merge path-patterns which occurs high computational cost. Recently, *Twig^{2}Stack* was proposed to overcome this problem using hierarchical-stacks to further reduce the merging cost. But, due to the complex hierarchical-stacks *Twig^{2}Stack* used, *Twig^{2}Stack* may end up many random accesses in memory, and need to load the whole XML tree into memory in the worst case. In this paper, we propose a new algorithm, called *TwigList*, which uses simple lists. Both time and space complexity of our algorithm are linear with respect to the total number of pattern occurrences and the size of XML tree. In addition, our algorithm can be easily modified as an external algorithm. We conducted extensive experimental studies using large benchmark and real datasets. Our algorithm significantly outperforms the up-to-date algorithm.

1 Introduction

The Extensible Markup Language (XML) is an emerging standard for data representation and exchange on the Internet. Pattern matching is one of the most important types of XML queries to retrieve information from an XML document. Among many reported studies, Zhang et al. in [1] introduced the region encoding to process XML queries and proposed a multi-predicate merge join algorithm using inverted list. Al-Khalifa et al. in [2] proposed a stack-based algorithm which breaks the twig query into a set of binary components. The drawback of the early work is the large intermediate results generated by the algorithm. Bruno et al. in [3] used a holistic twig join algorithm *TwigStack* to avoid producing large intermediate results. Jiang et al. in [4] proposed an XML Region Tree (*XR-tree*) which is a dynamic external memory index structure specially designed for nested XML data. With *XR-tree*, they presented a *TSGeneric+* algorithm to effectively skip both ancestors and descendants that do not participate in a join. Lu et al. in [5] proposed *TwigStackList* to better handle twig queries with parent-child relationships. Lu et al. in [6] used a different labeling scheme called extended Dewey, and proposed a *TJFast* algorithm to access only leaf elements. However, all of the above algorithms can not avoid a large number of unnecessary path mergings as theoretically shown in [7]. Hence, Aghili et al. in [8] proposed a binary labeling algorithm using

the method of nearest common ancestor to reduce search space. However, this technique is efficient in the cases when the returned nodes are the leaf nodes in the twig query. Most recently, Chen et al. in [9] proposed a *Twig^{2}Stack* algorithm which uses hierarchical-stacks instead of enumeration of path matches. *Twig^{2}Stack* outperforms *TwigStack* and *TJFast*. But *Twig^{2}Stack* may conduct many random accesses and may use a large memory space due to the complexity of hierarchical-stacks it uses.

The main contribution of this paper is summarized below. We present a new algorithm, called *TwigList*, which is independently developed and shares similarity with *Twig^{2}Stack* [9]. Our algorithm significantly outperforms *Twig^{2}Stack*. The efficiency of our *TwigList* algorithm is achieved by using simple lists rather than the hierarchical-stacks used in *Twig^{2}Stack* to reduce the computational cost. In addition, because of the simple list data structure and maximization of possible sequential scans used in our algorithm, we extend *TwigList* as an external algorithm, which still outperforms *Twig^{2}Stack* using a 582MB `XMark` benchmark, a 337MB `DBLP` dataset, and a 84MB `TreeBank` dataset, as reported in our extensive experimental studies.

The remainder of this paper is organized as follows. Section 2 gives the problem of processing twig-pattern matching queries. Section 3 discusses two existing algorithms and outlines their problems. We give our new algorithm in Section 4. Experimental results are presented in Section 5. Finally, Section 6 concludes the paper.

2 Twig-Pattern Matching Queries

An XML document can be modeled as a rooted, ordered, and node-labeled tree, $\mathcal{T}$, where a node represents an XML element, and an edge represents a parent/child relationship between elements in XML. For simplicity, in this work, a label of a node is a value that belongs to a type (tag-name). An example of an XML tree is shown in Fig. 1 (a). In the XML tree, a node is associated with a value x_i which belongs to a type X (denoted $x_i \in X$). For example, the root node has a value a_1 that belongs to type A. The ordering among sibling nodes specifies a traversal order.

A twig-pattern matching query is a fragment of XPATH queries that can be represented as a query tree, $Q(V, E)$. Here, $V = \{V_1, V_2, \cdots, V_n\}$ is a set of nodes representing types. We let V_i denote both the i^{th} typed query node in Q and the set of the i^{th} typed elements in the XML tree $\mathcal{T}$, E is a set of edges. An edge between two typed nodes, for example, A and D, is either associated with an XPATH axis operator $/\!/$ or $/$ to represent $A/\!/D$ or A/D. Given an XML tree $\mathcal{T}$, the former is to retrieve all A and D typed elements that satisfy the ancestor/descendant relationships, and the latter is to retrieve all A and D typed elements that satisfy parent/child relationships. We call the former $/\!/$-edge and the latter $/$-edge in short. As a special case, the root node has an incoming $/\!/$- or $/$-edge to represent an XPATH query, $/\!/A$ or $/A$, suppose the root node is A-typed. The answer of a n-node query tree, $Q(V, E)$, against an XML tree $\mathcal{T}$, is a set of all n-ary tuples $(v_1, v_2, \cdots, v_n)$ in $\mathcal{T}$, for $v_i \in V_i$ $(1 \leq i \leq n)$, that satisfy all the structural relationships imposed by Q. Consider an XPATH $Q = /\!/A[/\!/C]/\!/B$. The query tree is illustrated in Fig. 1 (b). When Q is issued against the XML tree (Fig. 1 (a)), the answer includes (a_1, b_1, c_3) and (a_3, b_3, c_1).

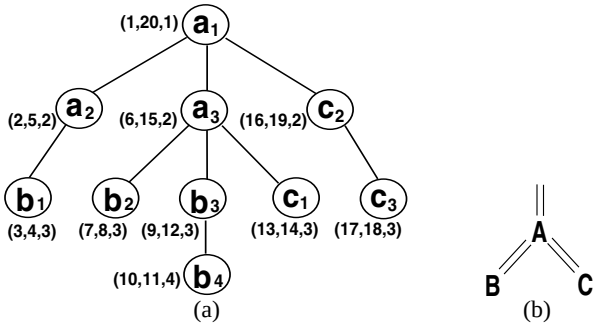


Fig. 1. An XML tree (a) and a query tree Q (b)

In this paper, we focus ourselves on efficient processing twig-pattern matching queries. For efficiently determining ancestor/descendant relationships among nodes in an XML tree, a node u is encoded with a triple, (s_u, e_u, d_u), where s_u and e_u together represent a region (starting/ending position), denoted $reg(u)$, and d_u represents the level of the node in the XML tree. (The root is at level 1.) With the region encoding (starting/ending position), a node u is an ancestor of a node v iff the $reg(v) \subseteq reg(u)$ such that $s_u < s_v \leq e_v < e_u$, a node u is a parent of a node v iff $reg(v) \subseteq reg(u)$ and $d_v = d_u + 1$, which implies that the node v is one level deeper than node u. For example, as shown in Fig. 1 (a), b_2 is a descendant of a_1, because the region of b_2, (7,8), is contained in the region of a_1 (1, 20). Also, b_2 is not a child of a_1 because the levels for b_2 and a_1 are 3 and 1.

3 Two Existing Algorithms: *TwigStack* And *Twig²Stack*

The twig-pattern matching query was first studied by Bruno, Koudas and Srivastava in [3]. A *TwigStack* algorithm was proposed to process a twig-pattern matching query, Q, in two steps. In the first step, in brief, a *PathStack* algorithm was proposed to efficiently process every query path in a given query tree. Consider the query $Q = /\!/A[/\!/C]/\!/B$ (Fig. 1 (b)). There are two query paths, $Q_{p_1} = /\!/A/\!/B$ and $Q_{p_2} = /\!/A/\!/C$. The *PathStack* algorithm finds answers for both of them using stacks. In the second step, *TwigStack* checks if the results for all the query paths can be merged to satisfy the structural relationships imposed by the given twig-pattern matching query. For *TwigStack*, the first step can be processed efficiently, but the second step consumes much time because it needs to process merging.

Below, in brief, we discuss the difficulties for *TwigStack* to reduce computational cost for merging in the second step after introducing *PathStack*. Consider a query path in the query tree $Q_p = /\!/V_1/\!/V_2/\!/ \cdots /\!/V_n$. A stack is created for every node V_i, denoted $stack(V_i)$. The whole query path is processed while traversing the given XML tree $\mathcal{T}$ following the preorder. When traversing a V_i-typed node v_i in XML tree $\mathcal{T}$ ($v_i \in V_i$), *PathStack* pops up nodes that are not ancestors of v_i in $stack(V_i)$ and $stack(V_{i-1})$, because they are no longer needed. Then *PathStack* pushes node v_i into $stack(V_i)$, iff

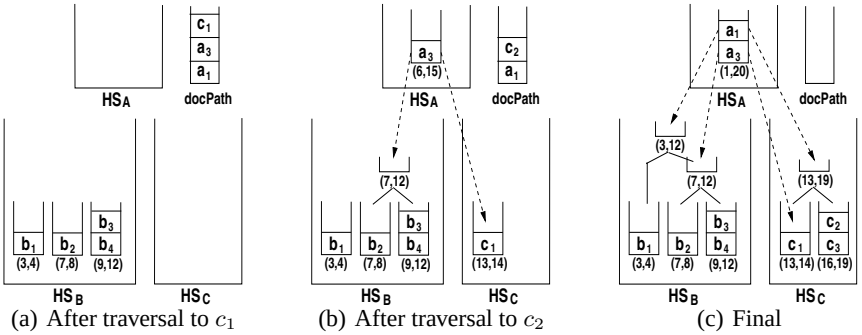
$stack(V_{i-1})$ is not empty. When v_i can be pushed into $stack(V_i)$, there is a pointer from v_i pointing to the top element in $stack(V_{i-1})$. Consider processing query path $Q_{p_1} = /\!/A/\!/B$ against XML tree $\mathcal{T}$ (Fig. 1 (a)). There are two stacks $stack(A)$ and $stack(B)$. Following preorder traversal, *PathStack* pushes a_1 and a_2 into $stack(A)$. When b_1 is traversed, the top element of a_2 in $stack(A)$ is an ancestor of b_1, so b_1 is pushed into $stack(B)$. *PathStack* will report (a_1, b_1) and (a_2, b_1) as result for the query path, because all the other elements in $stack(A)$ are ancestors of the top element. Then, a_3 is traversed, and *PathStack* will pop up a_2 before pushing a_3 into $stack(A)$, because a_2 is not an ancestor of a_3 and is not needed in the later processing. Similarly, when b_2 is traversed, b_1 is popped up. The merging process ensures the results satisfying the entire structural relationships. Reconsider $Q = /\!/A[/\!/C]/\!/B$ (Fig. 1 (b)) against XML tree $\mathcal{T}$ (Fig. 1 (a)). Here, (a_3, b_2) satisfies $Q_{p_1} = /\!/A/\!/B$, (a_1, c_2) satisfies $Q_{p_2} = /\!/A/\!/C$, but the two do not jointly satisfy $Q = /\!/A[/\!/C]/\!/B$. The cost of merging is considerably high as processing n joins, if there are n query paths for a twig-pattern matching query.

It is worth noting that *TwigStack* cannot allow the same stack, say $stack(A)$, to be shared by two query paths $Q_{p_1} = /\!/A/\!/B$ and $Q_{p_2} = /\!/A/\!/C$, and process twig-pattern matching queries without the merging step. It is because the sibling relationships cannot be easily maintained in the framework of *TwigStack*, and the push/popup, that are designed for each query path, cannot be used to control multiple paths (branches). Due to the different timing of push/popup, some answer may be missing.

In order to avoid the high cost in the step of merging, Chen et al. in [9] proposed a *Twig^{2}Stack* algorithm which instead uses a hierarchical-stack, denoted HS_{V_i}, for each node, in query tree Q, to compactly maintain all twig-patterns for a twig-pattern matching query.

Consider a query tree $Q(V, E)$ with n nodes ($V = \{V_1, V_2, \cdots, V_n\}$). *Twig^{2}Stack* maintains n hierarchical-stacks HS_{V_i} for $1 \leq i \leq n$. Each HS_{V_i} maintains an ordered sequence of stack-trees, $ST_1(V_i), ST_2(V_i), \cdots$, and a stack-tree, $ST_j(V_i)$, is an ordered tree of stacks. Each stack in the stack-tree contains zero or more document elements. The ancestor/descendant relationships are maintained by the stacks in the hierarchical-stacks. Suppose in an XML tree, u is an ancestor of v. If u and v have the same type, say V_i, in *Twig^{2}Stack*, they may appear in the same stack. If so, v will be pushed into the stack before u is in HS_{V_i}. If u and v have different types, V_i and V_j, then u will be in one stack in HS_{V_i} and v will be in one stack in HS_{V_j} and there is a pointer from the stack in HS_{V_i} to the stack in HS_{V_j} to represent their ancestor/descendant relationship.

Take an example of processing $Q = /\!/A[/\!/C]/\!/B$ (Fig. 1 (b)) against XML tree $\mathcal{T}$ (Fig. 1 (a)). *Twig^{2}Stack* traverses $\mathcal{T}$ in preorder: a_1, a_2, b_1, a_3, b_2, b_3, b_4, c_1, c_2, and c_3, and will push them into a special stack called $docpath$ in such order. Initially, a_1, a_2 and b_1 are pushed into the stack $docpath$ in order. When *Twig^{2}Stack* is about to push a_3 into $docpath$, it finds that b_1 is not an ancestor of a_3 and therefore pops-up b_1 from $docpath$ and pushes b_1 to the hierarchical-stack HS_B, and it then finds that a_2 is not an ancestor of a_3 either and therefore simply discards it (because a_2 does not have any C-typed descendants now, and will not have any later). When *Twig^{2}Stack* is about to push b_3 into $docpath$ after pushing b_2 into $docpath$, *Twig^{2}Stack* finds that b_2 is not b_3's ancestor, it pops up b_2 from $docpath$ and pushes b_2 into HS_B. Since b_2 is not an ancestor of b_1, there will be two single-node stack-trees in HS_B. Fig. 2 (a) shows



(a) After traversal to c_1 (b) After traversal to c_2 (c) Final

Fig. 2. *Twig*^{2}*Stack* for the query tree Q (Fig. 1 (b)) against XML tree $\mathcal{T}$ (Fig. 1 (a))

the $docpath$, the hierarchical-stacks after c_1 is pushed into $docpath$. Fig. 2 (b) shows the $docpath$, the hierarchical-stacks, HS_A and HS_B and HS_C, after c_2 is pushed into $docpath$. Note: there are two stack-trees in HS_B. From a_3 in HS_A, there is a pointer pointing to a subtree in HS_B indicating that it is an ancestor of b_2, b_3 and b_4; also there is a pointer to HS_C indicating that a_3 is an ancestor c_1. Fig. 2 (c) shows the hierarchical-stacks after all XML tree nodes are pushed/popped-up into/from $docpath$. As can be seen from Fig. 2 (c), all twig-patterns are maintained by the stacks in the hierarchical-stacks. After the hierarchical-stacks are constructed, *Twig*^{2}*Stack* enumerates the results in a bottom-top manner. For example, for a_1, *Twig*^{2}*Stack* enumerates the stacks, and conduct Cartesian-product between a_1 and $\{b_1, b_2, b_3, b_4\}$ and $\{c_1, c_2, c_3\}$.

As shown in [9], *Twig*^{2}*Stack* is a linear-time (w.r.t. the number of nodes of $\mathcal{T}$) algorithm to construct the hierarchical-stacks, and is a linear-time (w.r.t. the total number of matchings) enumeration algorithm based on intermediate structures maintained in the hierarchical-stacks. But there are also some problems in *Twig*^{2}*Stack*. First, the way of maintaining ancestor/descendant relationships across the hierarchical-stacks is too complex, which results in a large number of random memory accesses and therefore increases the processing time. Second, *TwigStack* needs to maintain a large number of stacks. In the worst case, it needs to load the whole XML tree into memory.

4 A New Algorithm: *TwigList*

We have developed a new algorithm *TwigList* for processing twig-pattern matching queries. The main difference between our *TwigList* algorithm and *Twig*^{2}*Stack* is that we do not need to maintain complicated hierarchical-stacks for nodes in a query tree Q. Instead of maintaining a hierarchical-stack for a node, V_i, in the query tree, *TwigList* simply maintains a list, L_{V_i}. The list is used based on the following remark.

Property 1. Consider $A/\!\!/B$ against an XML tree $\mathcal{T}$. If an A-typed node is an ancestor of a set of B-typed nodes in XML tree $\mathcal{T}$: (1) It must be able to specify a minimal interval for the A-typed node to cover all such B-typed nodes; (2) It must be the case that there does not exist any B-typed node in the interval that is not a descendant of the A-typed node.

Algorithm 1 *TwigList* $(Q, \mathcal{T})$

Input: a query tree Q with n nodes $\{V_1, \cdots, V_n\}$, and an XML tree $\mathcal{T}$;
Output: all n-ary tuples as answers for Q;

1: let X_i be a sequence of V_i-typed XML tree nodes sorted in preorder, for all $1 \leq i \leq n$;
2: let X be the set of all X_i for $1 \leq i \leq n$;
3: obtain a set of lists $L = \{L_{V_i}, L_{V_2}, \cdots L_{V_n}\}$ by calling *TwigList-Construct* (Q, X);
4: $R \leftarrow$ *TwigList-Enumerate* (Q, L);
5: **return** R;

It is important to know that existing algorithms *TwigStack* and *Twig*^{2}*Stack* do not fully make use of this property. Push/pop operations together with stacks cannot effectively maintain this property. We fully and effectively make use of this property. Unlike *TwigStack* and *Twig*^{2}*Stack*, we mainly use lists instead of stacks. Unlike *TwigStack*, we minimize the cost of enumerating results to the minimum (linear time), because the merging procedure of n joins is avoided. Unlike *Twig*^{2}*Stack*, we do not need to use complex hierarchical-stacks, and maximize the possibility to conduct sequential scans over the lists. When generating outputs for a twig-pattern matching query, Q, by enumerating the generated lists, we do not need to use any extra memory space, which further saves cost. Our algorithm is optimal in the sense that both time and space complexities of our algorithm are linear w.r.t. the total number of occurrences of twig-pattern matchings and the size of XML tree. As shown in our experimental studies, our external *TwigList* algorithm outperforms *Twig*^{2}*Stack* as well as *TwigStack*.

TwigList algorithm is outlined in Algorithm 1, which takes two inputs, a query tree $Q(V, E)$, representing a twig-pattern matching query, and an XML tree, $\mathcal{T}$. The query tree has n nodes, $\{V_1, V_2, \cdots, V_n\}$. *TwigList* constructs lists for all V_i-typed nodes in $\mathcal{T}$, and sorts them following preorder. There are two main steps. First, it calls *TwigList-Construct* to obtain a set of lists that compactly maintain all twig-patterns for answering Q (line 3). Second, it calls *TwigList-Enumerate* to obtain all n-ary tuples for Q (line 4). In the following, we discuss the two main algorithms, *TwigList-Construct* and *TwigList-Enumerate*, in detail. For simplicity, we first concentrate on query trees, Q, where only $/\!/$-edges appear. Then, we will discuss how to process a query tree with $/\!/$-edges as well as $/$-edges.

4.1 *TwigList-Construct* Algorithm

TwigList-Construct is outlined in Algorithm 2. We explain how *TwigList* works using an example of the same twig-pattern matching query $Q = /\!/A[/\!/C]/\!/B$ (Fig. 1 (b)) against XML tree $\mathcal{T}$ (Fig. 1 (a)). In Q, there are three types A, B, and C. A is the root node, and B and C are leaf nodes. Accordingly, as input, X consists of three sequences for the three types, $X_A = \langle a_1, a_2, a_3 \rangle$, $X_B = \langle b_1, b_2, b_3, b_4 \rangle$, and $X_C = \langle c_1, c_2, c_3 \rangle$. *TwigList-Construct* will generate three lists, L_A, L_B and L_C, to determine all possible n-ary tuples for answering Q. Here, for this Q, every XML tree node, a_i, in L_A will maintain two pairs of pointers to specify the intervals for its B-typed descendants, $(start_B, end_B)$, and its C-typed descendants, $(start_C, end_C)$.

Algorithm 2 *TwigList-Construct* (Q, X)

Input: a query tree Q with n nodes $\{V_1, \cdots, V_n\}$, and a set of sequences X, a sequence in X, X_i, maintains a list of V_i-typed XML nodes;
Output: all L_{V_i} for each $1 \leq i \leq n$.

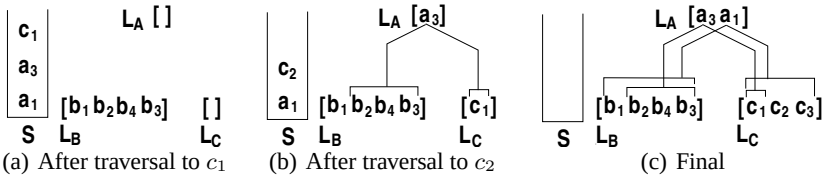
1: initialize stack S as empty;
2: initialize all list L_{V_i} as empty for all $V_i \in V(Q)$ (The length of L_{V_i} is initialized as 0);
3: **while** not all $X_q = \emptyset$, for $1 \leq q \leq n$, **do**
4: let V_q be the node such that its top element is the first following the preorder traversal among all top elements in all X_i;
5: let v be the top element in X_q;
6: remove v from X_q;
7: *toList*(S, Q, $reg(v)$);
8: **for each** child of V_q in query tree Q, V_p, **do**
9: $v.start_{V_p} \leftarrow length(L_{V_p}) + 1$;
10: push(S, v);
11: *toList*(S, Q, (∞, ∞));

Procedure *toList*(S, Q, r)

12: **while** $S \neq \emptyset \wedge r \not\subseteq reg(top(S))$ **do**
13: $v_j \leftarrow pop(S)$;
14: let v_j's type be V_j;
15: **for each** child of V_j in query tree Q, V_k, **do**
16: $v_j.end_{V_k} \leftarrow length(L_{V_k})$;
17: append v_j into list L_{V_j} if $v_j.start_{V_k} \leq v_j.end_{V_k}$ for every V_j's child, V_k;

Initially, it initializes a working stack S to be empty (line 1), and create empty lists, L_A, L_B, and L_C (line 2). Below, we use $length(L_X)$ to indicate the length of the list L_X. All the lengths of the lists are zero. In line 3-10, it repeats until all sequences, X_A, X_B, and X_C, become empty. In every iteration, *TwigList-Construct* selects a node from the sequences that is the first following preorder (line 4-6). For this example, *TwigList-Construct* access a_1, a_2, b_1, a_3, b_2, b_3, b_4, c_1, c_2, and c_3 in order, and will push them into S.

Suppose a_1, a_2 and b_1 are pushed into S already. a_1 and a_2's $start_B$ and $start_C$ pointers will point to the ends of L_B and L_C ($length(L_B) + 1$ and $length(L_C) + 1$), respectively. Their end_B and end_C will be updated later, because they are unknown now. When *TwigList-Construct* is about to push a_3 into S, it calls *toList* (line 7) with its region-code $reg(a_3)$. The body of *toList* is from line 12 to line 17. *toList* finds that b_1 as the top element in S is not an ancestor of a_3 and therefore pops-up b_1 from S and appends b_1 to L_B. Here, a_3's $start_B$ will point to $length(L_B) + 1$, because b_1 is not a descendant of a_3 and a_3's B-typed descendants will come after it, if any. a_3's $start_C$ will point to L_C ($length(L_C) + 1$) which is still empty. Then, *toList* also finds that a_2 as the current top element in S is not an ancestor of a_3. *toList* does not append it into L_A because it does not have any C-typed descendants now, and will not have any later (line 17). When *TwigList-Construct* is about to push b_3 into S after pushing b_2 into S, *toList* finds that b_2 is not b_3's ancestor, it pops up b_2 from S and appends b_2 to L_B.



(a) After traversal to c_1 (b) After traversal to c_2 (c) Final

Fig. 3. *TwigList-Construct* for the query tree Q (Fig. 1 (b)) against XML tree $\mathcal{T}$ (Fig. 1 (a))

Fig. 3 (a) shows the stack S and the lists, L_A, L_B, and L_C, after c_1 is pushed into S. Fig. 3 (b) shows S and the lists after c_2 is pushed into S. When c_2 is pushed into S, *toList* enforces c_1 and a_3 to be popped up, and append to the corresponding lists. This is the timing for a_3 to fill in its end_B and end_C positions ($length(L_B)$ and $length(L_C)$), respectively. Fig. 3 (c) shows the stack S and the lists after all XML tree nodes are pushed/popped-up into/from S. As can be seen from Fig. 3 (c), all twig-patterns are maintained by the lists.

In line 11, *TwigList-Construct* uses (∞, ∞) as the largest region code to enforce all in stack S to be appended into a list if possible.

Time/Space Complexity: Given a twig-pattern matching query, Q, and an XML tree $\mathcal{T}$. Suppose the corresponding query tree, Q, has n nodes, $V_1, V_2, \cdots, V_n$. The time/space complexity of *TwigList-Construct* (Algorithm 2) are both O($d \cdot |X|$) in the worst case, where $|X|$ is the total number of nodes, v_i, in XML tree that is V_i-typed $1 \leq i \leq n$, and d is the max degree of a node in the query tree Q. Note: every XML tree node that is V_i-typed will be pushed/popped-up into/from the stack S only once. It needs at most d times to calculate its intervals. Therefore, *TwigList-Construct* is linear w.r.t. $|X|$.

4.2 *TwigList-Enumerate* Algorithm

TwigList-Enumerate is outlined in Algorithm 3. It takes two input parameters, the n-node query tree Q and a set of lists, L, which consists of the lists obtained in *TwigList-Construct*. *TwigList-Enumerate* simply inserts all n-ary tuples as answers into a relation R to be returned.

Continue the above example, the three lists, L_A, L_B, and L_C, are shown in Fig. 3 (c). Initially, $start = [a_3, b_2, c_1]$, and $end = [a_1, b_3, c_1]$ (line 2-4). Here, pair (a_3, a_1) specifies the interval for all A-typed XML tree nodes. (b_2, b_3) specifies the interval where a_3's B-typed descendants exist, and (c_1, c_1) specifies the interval where a_3's C-typed descendants exist. Line 5, $move = [a_3, b_2, c_1]$ records the current positions for outputting results. Then, it calls *moreMatch* to generate all n-ary tuples. In *moreMatch*, it first inserts the n-ary tuple pointed by the n-element $move$ array. The termination condition is specified in line 9 when there is no tuple to be generated. *TwigList-Enumerate* will result (a_3, b_2, c_1), (a_3, b_4, c_1), and (a_3, b_3, c_1), followed by $(a_1, b_1, c_1), \cdots$.

Time/Space Complexity: Given a twig-pattern matching query, $Q(V, E)$, as a query tree, and an XML tree $\mathcal{T}$. Suppose lists $L_{V_1}, L_{V_2}, \cdots, L_{V_n}$ have been constructed, the time complexity of *TwigList-Enumerate* algorithm is $O(n \cdot |R|)$, where $|R|$ is the total number of twig-pattern matchings, and n is the number of nodes in query tree, Q.

Algorithm 3 *TwigList-Enumerate* (Q, L)

Input: a query tree, Q, with n nodes $\{V_1, \cdots, V_n\}$; a set of lists, L, consisting of all L_{V_i}, for $1 \leq i \leq n$;
Output: all n-ary tuples as answers for Q;

1: let $R \leftarrow \emptyset$;
2: let $start[1..n]$, $end[1..n]$ be n-element arrays for maintaining the regions for the n lists;
3: let V_1 be the root node in the query tree Q; $start[1]$ and $end[1]$ point to the begin and end positions of L_{V_1};
4: initialize the other $start[i]$ and $end[i]$ for V_i as the interval specified by the first element in its parent list;
5: let $move[1..n]$ be a n-element array where $move[i] \leftarrow start[i]$;
6: **while** *moreMatch*$(R, start, end, move) \neq$ false do;
7: **return** R;

Function *moreMatch*$(R, start, end, move)$

8: insert $(move[1], \cdots, move[n])$ as a n-ary tuple into R;
9: **if** $\forall i$: $move[i] = end[i]$ **then return** false;
10: select V_i such that $move[i] < end[i]$, but all its descendants, V_j, in the query tree Q, $move[j] = end[j]$;
11: $move[i] \leftarrow move[i] + 1$;
12: let v_i be the V_i-typed element pointed by $move[i]$;
13: **for** all V_i's descendants, V_j, in query tree Q **do**
14: reset all their $start[j]$, $end[j]$, and $move[j]$ according to the interval specified by its parent (rooted at v_i);
15: **return** true;

Because in each run of function *moreMatch*, we will get one more matching (line 8), and the operations below in *moreMatch* require time $O(n)$. The space complexity of *TwigList-Enumerate* is the same for *TwigList-Construct*, because it does not consume any more memory space, other than three arrays $start[1..n]$, $end[1..n]$, and $move[1..n]$. Hence, the time complexity for *TwigList* is the sum of that for *TwigList-Construct* and *TwigList-Enumerate*, $O(d \cdot |X| + n \cdot |R|)$. This algorithm is optimal because it is linear w.r.t. $|R|$ and $|X|$. Note $O(n \cdot |R|)$ is lower bound to output all twig-pattern matchings of a n-node query tree explicitly.

4.3 Discussions

Handling $/$-Edges in Query Trees: If there are $/$-edges in a query tree, Q, it needs to have additional information to maintain the sibling information for efficiently processing twig-pattern matching queries. Consider $Q' = //A[//C]/B$ against the XML tree $\mathcal{T}$ (Fig. 1 (a)). Only (a_3, b_2, c_1) and (a_3, b_3, c_1) are the answers of Q'. We can simply extend *TwigList-Construct* to construct lists when there are $/$-edges in a given query tree. For $Q' = //A[//C]/B$, the lists constructed are shown in Fig. 4. As shown in Fig. 4, there is no need to append a_1 into list L_A, because a_1 does not have a B-typed child when it is about to append. When b_3 is about to be appended into L_B, *TwigList-Construct* knows that b_2 is a sibling which shares the same A-typed parent of b_3, a

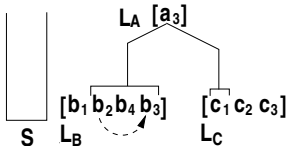


Fig. 4. *TwigList-Construct* for $Q' = /\!\!/A[/\!\!/C]/B$ against XML tree $\mathcal{T}$ (Fig. 1 (a))

sibling link can be added from b_2 to b_3. Note: some b_i and c_j are not in the interval of a_3 as shown in Fig. 4, it is because that it is unknown whether it is in the interval of its parent when it is appended into the corresponding list. With the sibling pointers, *TwigList-Enumerate* can quickly enumerate all results.

External Algorithm: When the set of lists L is too large to fit into memory, *TwigList* can be simply extended to work as an external algorithm by maintaining all lists on disk. It is because the access patterns against the lists usually focus on intervals and are not random. We implemented an external *TwigList* algorithm with n 4KB-pages for a query tree with n nodes. The external *TwigList* algorithm outperforms *Twig^{2}Stack*.

5 Performance Study

We have implemented three algorithms for processing twig-pattern matching queries: *Twig^{2}Stack* [9], our *TwigList*, and our external version of *TwigList* (*E-TwigList*) using C++. We choose *Twig^{2}Stack* as the basis to compare, because *Twig^{2}Stack* is the most up-to-date algorithm which outperforms *TwigStack* [3] and *TJFast* [6]. *TJFast* is a fast algorithm for processing twig-pattern matching queries with both $/\!\!/$-edges and $/$-edges.

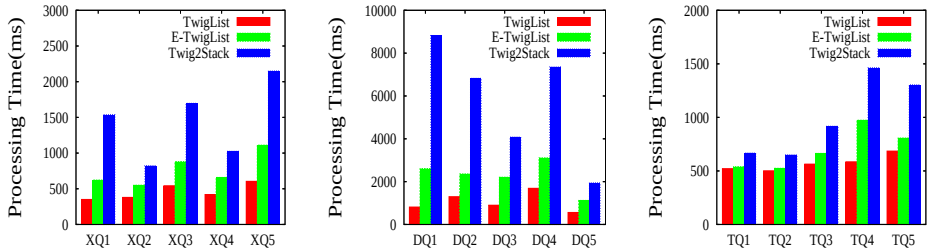
Three Datasets: We used both benchmark dataset, `XMark`, and two real datasets, `DBLP` and `TreeBank`. For `XMark`, we set the scaling factor to be 5.0 and generated a 582MB `XMark` dataset with 77 different labels and a maximum depth of 12. For real datasets, we use a 337MB `DBLP` dataset which has 41 different labels and a maximum depth of 6, and the 84MB `TreeBank` dataset which has 250 different labels and a maximum depth of 36. The `DBLP` dataset is wide but shallow, whereas the `TreeBank` dataset is deep and recursive.

All experiments were performed on a 2.8G HZ Pentium (R)4 processor PC with 1GB RAM running on Windows XP system. We mainly report processing time for construction and enumeration used in *Twig^{2}Stack*, *TwigList*, and *E-TwigList*, since the other time as loading and storing final results are the same. The buffer size used for *E-TwigList* is a 4KB-page for every node in a query tree.

Twig-pattern matching queries: We conducted extensive testing, and report the results for 15 twig-pattern matching queries (query trees) as shown in Table 1. For each of the three datasets, we report five query trees, which have different combinations of $/$-edges and $/\!\!/$-edges and different selectivities. In each group of 5 query trees, the first 2 are selected from the queries used in [9], and the second 2 are constructed by adding some branches into the first 2. The last is a rather complex query tree.

Name	Dataset	QueryTrees	ResultSize
XQ1	XMark	//item[location]/description//keyword	136,282
XQ2	XMark	//people//person[.//address/zipcode]/profile/education	15,859
XQ3	XMark	//item[location][.//mailbox/mail//emph]/description//keyword	86,533
XQ4	XMark	//people//person[.//address/zipcode][id]/profile[.//age]/education	7,997
XQ5	XMark	//open_auction[.//annotation[.//person]//parlist]//bidder//increase	141,851
DQ1	DBLP	//dblp/inproceedings[title]/author	1,205,196
DQ2	DBLP	//dblp/article[author][.//title]//year	625,991
DQ3	DBLP	//dblp/inproceedings[.//cite/label][title]/author	132,902
DQ4	DBLP	//dblp/article[author][.//title][.//url][.//ee]//year	384,474
DQ5	DBLP	//article[.//mdate][.//volume][.//cite//label]//journal	13,785
TQ1	TreeBank	//S/VP//PP[.//NP/VBN]/IN	1,183
TQ2	TreeBank	//S/VP/PP[IN]/NP/VBN	152
TQ3	TreeBank	//S/VP//PP[.//NN][.//NP[.//CD]/VBN]/IN	381
TQ4	TreeBank	//S[.//VP][.//NP]/VP/PP[IN]/NP/VBN	1,185
TQ5	TreeBank	//EMPTY[.//VP/PP//NNP][.//S[.//PP//JJ]//VBN]//PP/NP//_NONE_	94,535

Table 1. A list of query trees used in the experiments



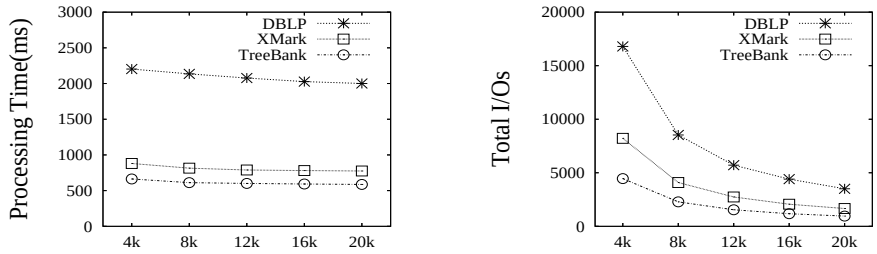
(a) XMark (b) DBLP (c) TreeBank

Fig. 5. Processing Time (ms)

Fig. 5 depicts the processing time of query trees listed in Table 1 for the datasets, XMark (Fig. 5 (a)), DBLP (Fig. 5 (b)), and TreeBank (Fig. 5 (c)). *TwigList* and even *E-TwigList* outperform *Twig*^{2}*Stack* in all tests. *TwigList* (*E-TwigList*) outperforms *Twig*^{2}*Stack*, mainly due to the linear structure (lists) used to organize the elements instead of complex hierarchical-stacks used in *Twig*^{2}*Stack*. Also, when enumerating results, *Twig*^{2}*Stack* uses a join approach which produces a lot of intermediate results, whereas our *TwigList* (*E-TwigList*) does not generate any intermediate results.

For XMark (Fig. 5 (a)), on average, *TwigList* is 3-4 times and *E-TwigList* is 2-3 times faster than *Twig*^{2}*Stack*. For DBLP (Fig. 5 (b)), on average, *TwigList* is 4-8 times and *E-TwigList* is 2-4 times faster than *Twig*^{2}*Stack*. For TreeBank (Fig. 5 (c)), *TwigList* and *E-TwigList* outperform *Twig*^{2}*Stack*, in particular when the query tree becomes complex, for example, TQ5. Our algorithms based on linear structures (lists) replace a large number of random accesses with sequential accesses in both memory and disk.

E-TwigList Test: We further test *E-TwigList* by choosing three queries, XQ3, DQ3 and TQ3, as representations of the queries over XMark, DBLP and TreeBank datasets. Their structures are moderately complex, and they produce a moderate number of matchings. XQ3 has 7 nodes with a tree of depth 4 and max node degree 3, DQ3 has 6 nodes with a tree of depth 4 and max degree 3, and TQ3 has 8 nodes with a tree of depth 5 and max degree 3. The total number of I/Os include the I/O cost in loading, construction and enumeration. We vary the buffer size from 4KB to 20KB. As shown in Fig. 6(b), we can see that the total number of I/Os decreases when the buffer size increases. We can



(a) Processing time varying buffer size (b) Total I/Os varying buffer size

Fig. 6. Total Processing time and I/Os varying buffer size

also see from Fig. 6(a) that the processing time decreases when the buffer size increases, but the effect of buffer sizes on processing times is not obvious. It concludes that only a small buffer is needed for a node in a query tree.

6 Conclusion and Future Work

In this paper, we propose a new *TwigList* algorithm to compactly maintain the twig-patterns using simple lists. The algorithm can be easily extended as an external algorithm (*E-TwigList*). The time and space complexity of the algorithm are linear with respect to the total number of occurrences of twig-patterns and the size of XML tree. Our algorithm significantly outperforms *Twig*^{2}*Stack* algorithm.

As the future work, we are planning to study the pattern matching over directed acyclic graphs using the similar method to get a better performance.

Acknowledgment: This work was supported by a grant of RGC, Hong Kong SAR, China (No. 418206).

References

1. Zhang, C., Naughton, J.F., DeWitt, D.J., Luo, Q., Lohman, G.M.: On supporting containment queries in relational database management systems. In: SIGMOD. (2001)
2. Al-Khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D.: Structural joins: A primitive for efficient XML query pattern matching. In: ICDE. (2002)
3. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD. (2002)
4. Jiang, H., Lu, H., Wang, W., Ooi, B.C.: Xr-tree: Indexing XML data for efficient structural joins. In: ICDE. (2003)
5. Lu, J., Chen, T., Ling, T.W.: Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In: CIKM. (2004)
6. Lu, J., Ling, T.W., Chan, C.Y., Chen, T.: From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In: VLDB. (2005)
7. Choi, B., Mahoui, M., Wood, D.: On the optimality of holistic algorithms for twig queries. In: DEXA. (2003)
8. Aghili, S., Li, H.G., Agrawal, D., Abbadi, A.E.: Twix: Twig structure and content matching of selective queries using binary labeling. In: INFOSCALE. (2006)
9. Chen, S., Li, H.G., Tatemura, J., Hsiung, W.P., Agrawal, D., Candan, K.S.: Twig2stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In: VLDB. (2006)